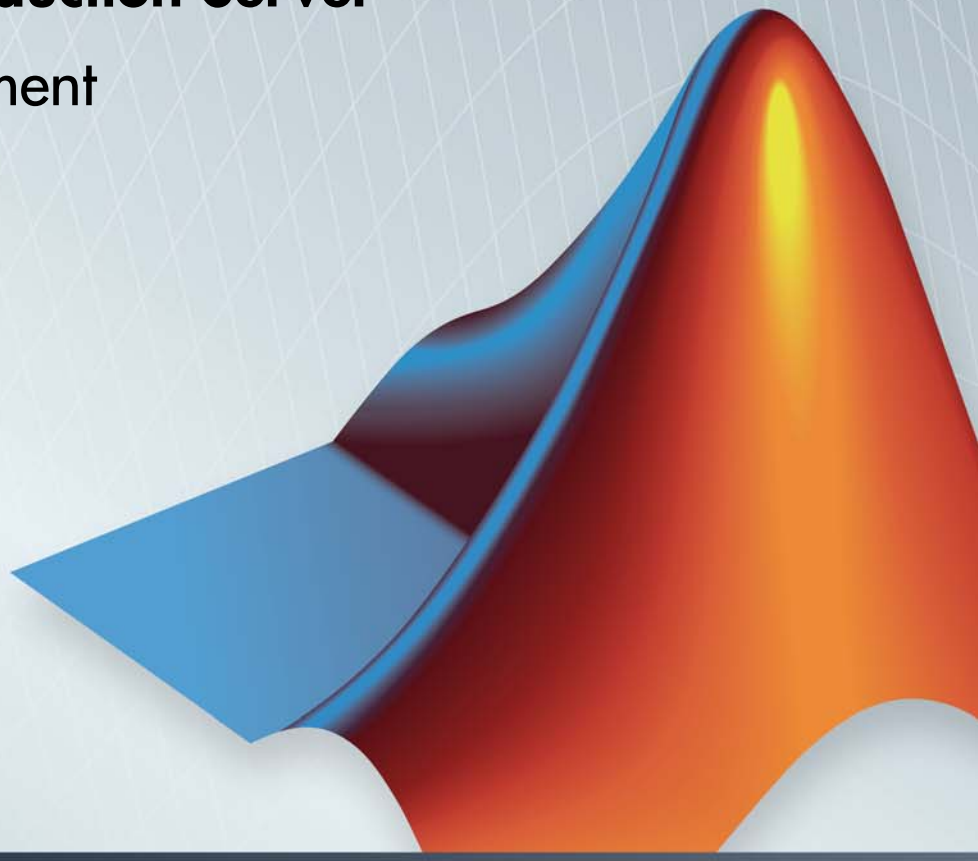
MATLAB® Production Server™

Code Deployment

R2014**a**



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Production Server™ Code Deployment

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

March 2014	Online only	New for Version 1.2 (Release R2014a)

Contents

1 Write Deployable MATLAB Code

Deployment Coding Guidelines 1-2

State-Dependent Functions 1-3
Does My MATLAB Function Carry State? 1-3
Defensive Coding Practices 1-3
Techniques for Preserving State 1-4

Deploying MATLAB Functions Containing MEX Files 1-6

Unsupported MATLAB Data Types for Client and Server Marshaling 1-7

2 Create a Deployable Archive from MATLAB Code

Compile a Deployable Archive with the Production Server Compiler App 2-2

Compile a Deployable Archive from the Command Line 2-8
Execute Compiler Projects with deploytool 2-8
Compile a Deployable Archive with mcc 2-8

Modifying Deployed Functions 2-10

3 Customizing a Compiler Project

Customize the Installer 3-2
- Change the Application Icon 3-2
- Add Application Information 3-3
- Change the Splash Screen 3-4
- Change the Installation Path 3-4
- Change the Logo 3-5
- Edit the Installation Notes 3-5

Manage Required Files in a Compiler Project 3-6
- Dependency Analysis 3-6
- Using the Compiler Apps 3-6
- Using mcc 3-7

Specify Files to Install with the Application 3-8

Manage Support Packages 3-9

4 Functions

1

Write Deployable MATLAB Code

- “Deployment Coding Guidelines ” on page 1-2
- “State-Dependent Functions” on page 1-3
- “Deploying MATLAB Functions Containing MEX Files” on page 1-6
- “Unsupported MATLAB Data Types for Client and Server Marshaling” on page 1-7

Deployment Coding Guidelines

MATLAB coding guidelines are essentially the same for both the deployment products and MATLAB® Production Server™ with important distinctions regarding functions that depend on MATLAB state.

Functions you deploy with MATLAB Production Server cannot be assumed to retain access to the same instance of the MATLAB Compiler Runtime, since the workers can access a number of different MCR instances. Therefore, when using MATLAB Production Server you must take extra care to ensure that state has not been changed or invalidated. See “State-Dependent Functions” on page 1-3 for more information.

Refer to “Write Deployable MATLAB Code” in the MATLAB Compiler™ documentation for general guidelines about deploying MATLAB code.

State-Dependent Functions

MATLAB code that you want to deploy often carries *state*—a specific data value in a program or program variable.

Does My MATLAB Function Carry State?

Example of carrying state in a MATLAB program include, but are not limited to:

- Modifying or relying on the MATLAB path and the Java® class path
- Accessing MATLAB state that is inherently persistent or global. Some example of this include:
  - Random number seeds
  - Handle Graphics® root objects that retain data
  - MATLAB or MATLAB toolbox settings and preferences
- Creating global and persistent variables.
- Loading MATLAB objects (MATLAB classes) into MATLAB. If you access a MATLAB object in any way, it loads into MATLAB.
- Calling MEX files, Java methods, or C# methods containing static variables.

Defensive Coding Practices

If your MATLAB function not only carries state, but *relies on it* for your function to properly execute, you must take additional steps (listed in this section) to ensure state retention.

When you deploy your application, consider cases where you carry state, and safeguard against that state's corruption if needed. *Assume* that your state may be changed and code defensively against that condition.

The following are examples of "defensive coding" practices:

Reset System-Generated Values in the Deployed Application

If you are using a random number seed, for example, reset it in your deployed application program to ensure the integrity of your original MATLAB function.

Validate Global or Persistent Variable Values

If you must use global or persistent variables, always validate their value in your deployed application and reset if needed.

Ensure Access to Data Caches

If your function relies on cached transaction replies, for instance, ensure your deployed system and application has access to that cache outside of the MATLAB environment.

Use Simple Data Types When Possible

Simple data types are usually not tied to a specific application and means of storing state. Your options for choosing an appropriate state-preserving tool increase as your data types become less complicated and specific.

Avoid Using MATLAB Callback Functions

Avoid using MATLAB callbacks, such as `timer`. Callback functions have the ability to interrupt and override the current state of the MATLAB Production Server worker and may yield unpredictable results in multiuser environments.

Techniques for Preserving State

The most appropriate method for preserving state depends largely on the type of data you need to save.

- Databases provide the most versatile and scalable means for retaining stateful data. The database acts as a generic repository and can generally work with any application in an enterprise development environment. It does not impose requirements or restrictions on the data structure or layout. Another related technique is to use comma-delimited files, in applications such as Microsoft® Excel®.
- Data that is specific to a third-party programming language, such as Java and C#, can be retained using a number of techniques. Consult the online documentation for the appropriate third-party vendor for best practices on preserving state.

Caution Using MATLAB `LOAD` and `SAVE` functions is often used to preserve state in MATLAB applications and workspaces. While this may be successful in some circumstances, it is highly recommended that the data be validated and reset if needed, if not stored in a generic repository such as a database.

Deploying MATLAB Functions Containing MEX Files

If the MATLAB function you are deploying uses MEX files, ensure that the system running MATLAB Production Server is running the version of MATLAB Compiler used to create the MEX files.

Coordinate with your server administrator and application developer as needed.

Unsupported MATLAB Data Types for Client and Server Marshaling

These data types are not supported for marshaling between MATLAB Production Server server instances and clients:

- MATLAB function handles
- Complex (imaginary) data
- Sparse arrays

2

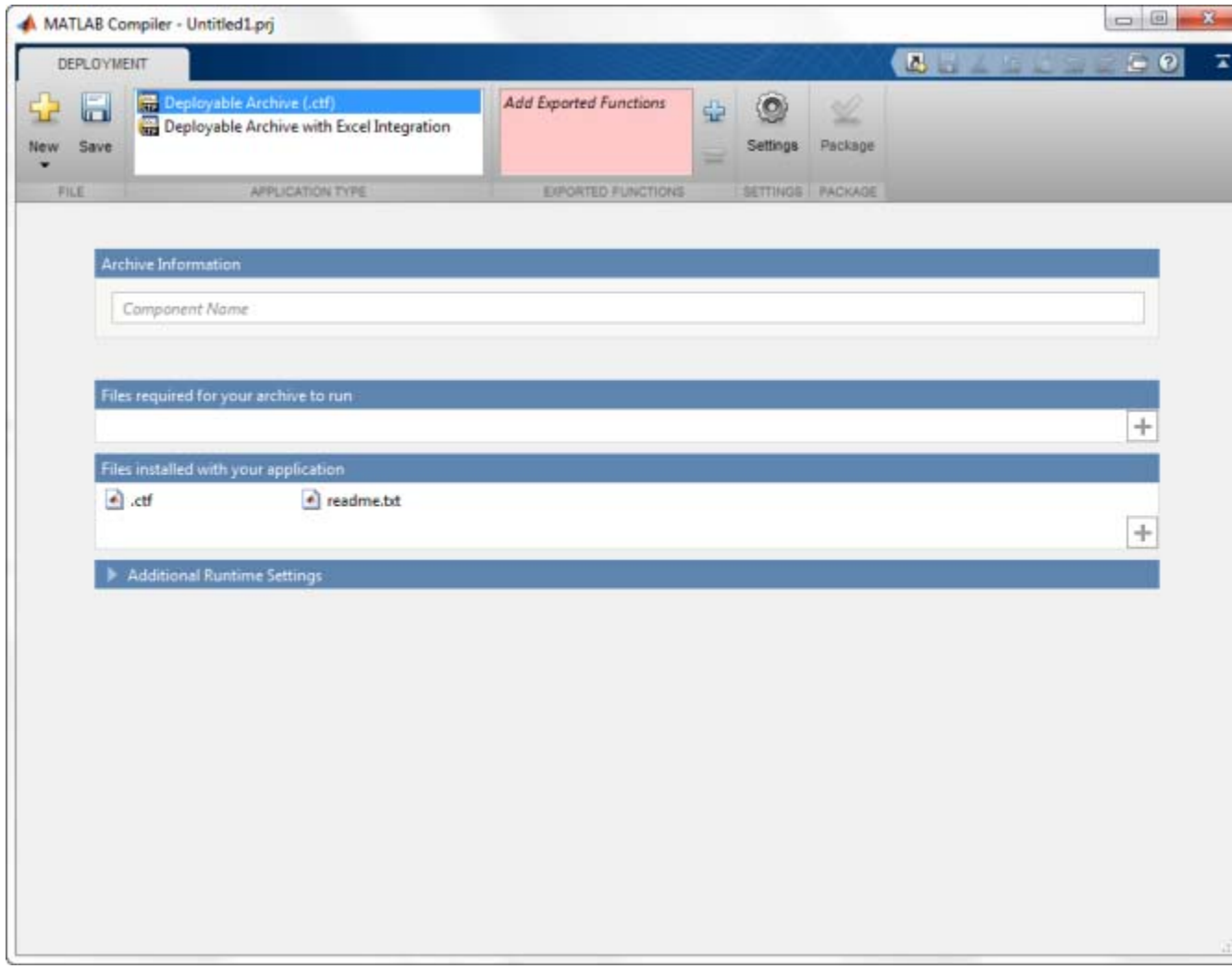
Create a Deployable Archive from MATLAB Code

- "Compile a Deployable Archive with the Production Server Compiler App" on page 2-2
- "Compile a Deployable Archive from the Command Line" on page 2-8
- "Modifying Deployed Functions" on page 2-10

Compile a Deployable Archive with the Production Server Compiler App

To compile MATLAB code into a deployable archive:

1. Open the **Production Server Compiler**.
   a. On the toolstrip select the **Apps** tab on the toolstrip.
   b. Click the arrow at the far right of the tab to open the apps gallery.
   c. Click **Production Server Compiler**.



Note To open an existing project, select it from the MATLAB **Current Folder** panel.

Note You can also launch the compiler using the `productionServerCompiler` function.

2. In the **Application Type** section of the toolstrip, select **Deployable Archive**.

Note If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

3. Specify the MATLAB files you want deployed in the package.
   a. In the **Exported Functions** section of the toolstrip, click the plus button.

   Note If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

   b. In the file explorer that opens, locate and select one or more the MATLAB files.
   c. Click **Open** to select the file and close the file explorer.

      The names of the selected files are added to the list and a minus button appears below the plus button. The name of the first file listed is used as the default application name.
4. In the **Packaging Options** section of the toolstrip, specify how the installer will deliver the MATLAB Compiler Runtime (MCR) with the archive.

Note If the **Packaging Options** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

You can select one or both of the following options:

- **Runtime downloaded from web** — Generates an installer that downloads the MCR installer from the Web.
- **Runtime included in package** — Generates an installer that includes the MCR installer.

Note Selecting both options creates two installers.

Regardless of the options selected the generated installer scans the target system to determine if there is an existing installation of the appropriate MCR. If there is not, the installer installs the MCR.

5. Specify the name of any generated installers.

6. In the **Application Information** and **Additional Installer Options** sections of the compiler, customize the look and feel of the generated installer.

   You can change the information used to identify the application data used by the installer:

   - Splash screen
   - Installer icon
   - Version
   - Name and contact information of the archive's author
   - Brief summary of the archive's purpose
   - Detailed description of the archive

   You can also change the default location into which the archive is installed and provide some notes to the installer.

   All of the provided information is displayed as the installer runs.

   For more information see "Customize the Installer" on page 3-2.

7 In the **Files required for your application to run** section of the compiler, verify that all of the files required by the deployed MATLAB functions are listed.

Note These files are compiled into the generated binaries along with the exported files.

The built-in dependency checker will automatically populate this section with the appropriate files. However, if needed you can manually add any files it missed.

For more information see "Manage Required Files in a Compiler Project" on page 3-6.

8 In the **Files installed with your application** section of the compiler, verify that any additional non-MATLAB files you want installed with the application are listed.

Note These files are placed in the `applications` folder of the installation.

This section automatically lists:

- Generated deployable archive
- Readme file

You can manually add files to the list. Additional files can include documentation, sample data files, and examples to accompany the application.

For more information see "Specify Files to Install with the Application" on page 3-8.

9 Click the **Settings** button to customize the flags passed to the compiler and the folders to which the generated files are written.

10 Click the **Package** button to compile the MATLAB code and generate any installers.

11 Verify that the generated output contains:

- `for_redistribution` — A folder containing the installer to distribute the archive
- `for_testing` — A folder containing the raw generated files to create the installer
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the archive
- `PackagingLog.txt` — A log file generated by the compiler

Compile a Deployable Archive from the Command Line

In this section...
"Execute Compiler Projects with deploytool" on page 2-8
"Compile a Deployable Archive with mcc" on page 2-8

You can compile deployable archives from both the MATLAB command line and the system terminal command line:

- `deploytool` invokes the compiler app to execute a pre-saved compiler project
- `mcc` invokes the raw compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke the compiler without opening a window:

- `-build` *`project_name`* — Invoke the compiler to build the project and do not generate an installer.
- `-package` *`project_name`* — Invoke the compiler to build the project and generate an installer.

For example, `deploytool -package magicsqaure` generates of the binary files defined by the `magicaqure` project and packages them into an installer that you can distribute to others.

Compile a Deployable Archive with mcc

The `mcc` command invokes the raw compiler and provides fine-level control over the compilation of the deployable archive. It, however, cannot package the results in an installer.

To invoke the compiler to generate a deployable arcive use the `-W CTF:`*`component_name`* flag with `mcc`. The `-W CTF:`*`component_name`* flag creates a deployable archive called *`component_name`*`.ctf`.

For compiling deployable archives, you can also use the following options.

Compiler Java Options

Option	Description
`-a` *`filePath`*	Add any files on the path to the generated binary.
`-d` *`outFolder`*	Specify the folder into which the results of compilation are written.
`class{`*`className`*`:`*`mfilename`*`...}`	Specify that an additional class is generated that includes methods for the listed MATLAB files.

Modifying Deployed Functions

Once you have built a deployable archive, you can modify your MATLAB code, recompile, and see the change instantly reflected in the archive hosted on your server. This is known as “hot deploying” or “redeploying” a function.

To Hot Deploy, you must have a server created and running, with the built deployable archive located in the server’s `auto_deploy` folder.

The server deploys the updated version of your archive when on the following occurs:

- Compiled archive has an updated time stamp
- Change has occurred to the archive contents (new file or deleted file)

It takes a maximum of five seconds to redeploy a function using Hot Deployment. It takes a maximum of ten seconds to undeploy a function (remove the function from being hosted).

3

Customizing a Compiler Project

- "Customize the Installer" on page 3-2
- "Manage Required Files in a Compiler Project" on page 3-6
- "Specify Files to Install with the Application" on page 3-8
- "Manage Support Packages" on page 3-9

Customize the Installer

In this section...
“Change the Application Icon” on page 3-2
“Add Application Information” on page 3-3
“Change the Splash Screen” on page 3-4
“Change the Installation Path” on page 3-4
“Change the Logo” on page 3-5
“Edit the Installation Notes” on page 3-5

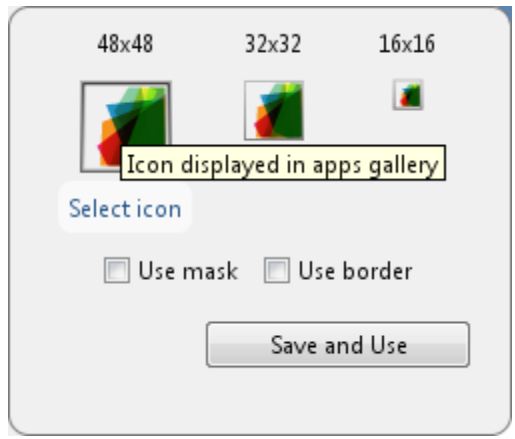
Change the Application Icon

The application icon is used for the generated installer. For standalone applications, it is also the application’s icon.

You can change the default icon in **Application Information**. To set a custom icon:

1 Click the graphic to the left of the **Application name** field.

A window previewing the icon opens.



2 Click **Select icon**.

3 Using the file explorer, locate the graphic file to use as the application icon.

4 Select the graphic file.

5 Click **OK** to return to the icon preview.

6 Select **Use mask** to fill any blank spaces around the icon with white.

7 Select **Use border** to add a border around the icon.

8 Click **Save and Use** to return to the main compiler window.

Add Application Information

The **Application Information** section of the compiler app allows you to provide these values:

- Name

  Determines the name of the installed MATLAB components. For example, if the name is `foo`, the installed executable would be `foo.exe`, the Windows® start menu entry would be **foo**. The folder created for the application would be *`InstallRoot`*`/foo`.

  The default value is the name of the first function listed in the **Main File(s)** field of the compiler.

- Version

  The default value is 1.0.

- Author name
- Support e-mail address
- Company name

  Determines the full installation path for the installed MATLAB components. For example, if the company name is `bar`, the full installation path would be *`InstallRoot`*`/bar/`*`ApplicationName`*.

- Summary
- Description

This information is all optional and, unless otherwise stated, is only used for display purposes. It appears on the first page of the installer. On Windows systems, this information is also displayed in the Windows **Add/Remove Programs** control panel.

Change the Splash Screen

The installer's splash screen displays after the installer is started. It is displayed, along with a status bar, while the installer initializes.

You can change the default image by clicking the **Select custom splash screen** link in **Application Information**. When the file explorer opens, locate and select a new image.

Note You can drag and drop a custom image onto the default splash screen.

Change the Installation Path

Default Installation Paths on page 3-4 lists the default path the installer will use when installing the compiled binaries onto a target system.

Default Installation Paths

Windows	`C:\Program Files\`*`companyName`*`\`*`appName`*
Mac OS X	`/Applications/`*`companyName`*`/`*`appName`*
Linux®	`/usr/`*`companyName`*`/`*`appName`*

You can change the default installation path by editing the **Default installation folder** field under **Additional Installer Options**.

The **Default installation folder** field has two parts:

- root folder — A drop down list that offers options for where the install folder is installed. Custom Installation Roots on page 3-5 lists the optional root folders for each platform.

Custom Installation Roots

Windows	`C:\Users\`*`userName`*`\AppData`
Linux	`/usr/local`

- install folder — A text field specifying the path appended to the root folder.

Change the Logo

The logo displays after the installer is started. It is displayed on the right side of the installer.

You change the default image by clicking the **Select custom logo** link in **Additional Installer Options**. When the file explorer opens, locate and select a new image.

Note You can drag and drop a custom image onto the default logo.

Edit the Installation Notes

Installation notes are displayed once the installer has successfully installed the packaged files on the target system. They can provide useful information concerning any additional set up that is required to use the installed binaries or simply provide instructions for how to run the application.

The field for editing the installation notes is in **Additional Installer Options**.

Manage Required Files in a Compiler Project

In this section...

"Dependency Analysis" on page 3-6

"Using the Compiler Apps" on page 3-6

"Using mcc" on page 3-7

Dependency Analysis

The compiler uses a dependency analysis function to automatically determine what additional MATLAB files are required for the application to compile and run. These files are automatically compiled into the generated binary. The compiler does not generate any wrapper code allowing direct access to the functions defined by the required files.

Using the Compiler Apps

If you are using one of the compiler apps, the required files discovered by the dependency analysis function are listed in the **Files required by your application to run** field.

To add files:

1. Click the plus button in the field.
2. Select the desired file from the file explorer.
3. Click **OK**.

To remove files:

1. Select the desired file.
2. Press the **Delete** key.

Caution Removing files from the list of required files may cause your application to not compile or to not run properly when deployed.

Using mcc

If you are using `mcc` to compile your MATLAB code, the compiler does not display a list of required files before running. Instead, it compiles all of the required files that are discovered by the dependency analysis function and adds them to the generated binary file.

You can add files to the list by passing one, or more, `-a` arguments to `mcc`. The `-a` arguments add the specified files to the list of files to be added into the generated binary. For example, `-a hello.m` adds the file `hello.m` to the list of required files and `-a ./foo` adds all of the files in `foo`, and its subfolders, to the list of required files.

Specify Files to Install with the Application

The compiler apps package files to install along with the ones it generates. By default the installer includes a readme file with instructions on installing the MATLAB Compiler Runtime(MCR) and configuring it.

These files are listed in the **Files installed with your application** section of the app.

to add files to the list:

1. Click the plus button in the field.
2. Select the desired file from the file explorer.
3. Click **OK** to close the file explorer.

To remove files from the list:

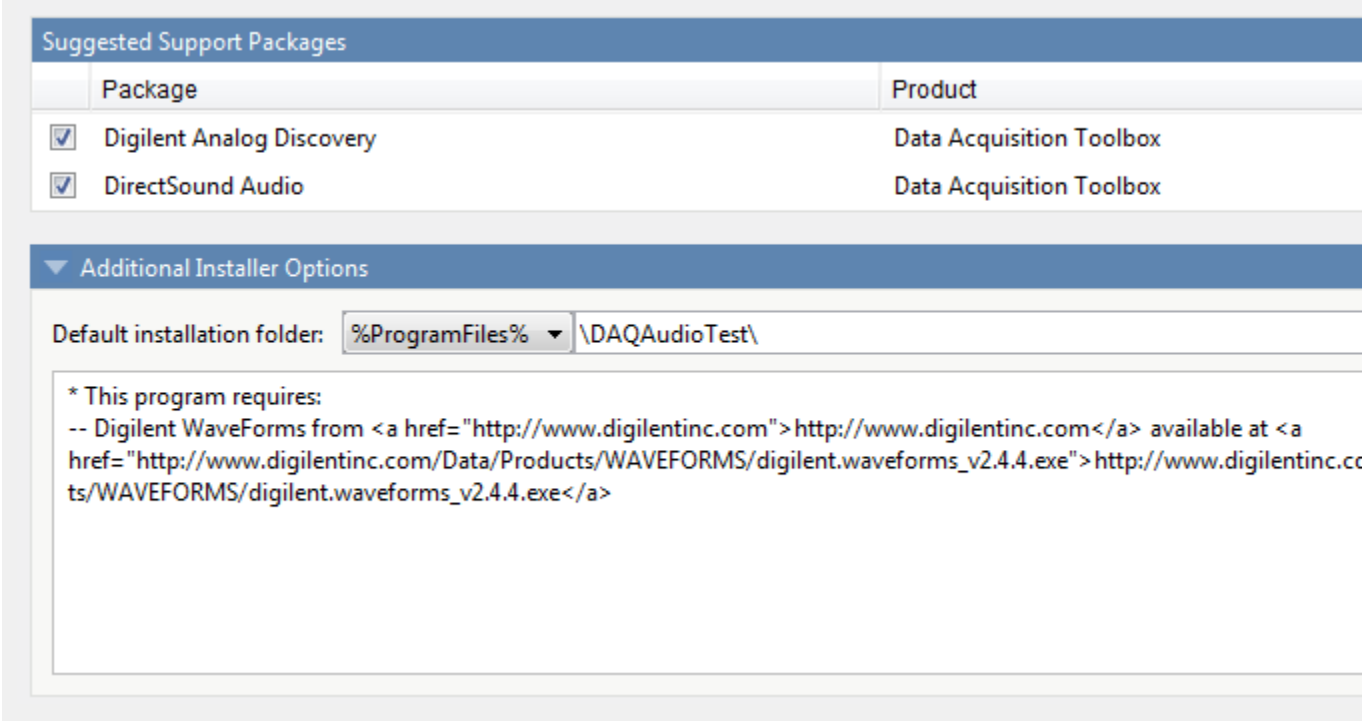
1. Select the desired file.
2. Press the **Delete** key.

Caution Removing the binary targets from the list results in an installer that does not install the intended functionality.

When installed on a target computer, the files listed in the **Files installed with your application** are placed in the `application` folder.

Manage Support Packages

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, MATLAB Compiler displays a **Suggested Support Packages** section.



The list displays all installed support packages that your MATLAB code requires. The list is determined using these criteria:

- the support package is installed
- your code has a direct dependency on the support package
- your code is dependent on the base product of the support package

- your code is dependent on at least one of the files listed as a dependency in the `mcc.xml` file of the support package, and the base product of the support package is MATLAB

Deselect support packages that are not required by your application.

Some support packages require third-party drivers that MATLAB Compiler cannot package. In this case, the compiler adds the information to the installation notes. You can edit installation notes in the **Additional Installer Options** section of the app. To remove the installation note text, deselect the support package with the third-party dependency.

Caution Any text you enter beneath the generated text will be lost if you deselect the support package.

4

Functions

productionServerCompiler

Purpose

Build and package functions for use with MATLAB Production Server

Syntax

`productionServerCompiler` [-win32] [[[-build] | [-project]]*`project_name`*]

Description

`productionServerCompiler` opens the MATLAB compiler for the creation of a new compiler project

`productionServerCompiler project_name` opens the MATLAB compiler with the project preloaded.

`productionServerCompiler -build project_name` runs the MATLAB compiler to build the specified project. The installer is not generated.

`productionServerCompiler -package project_name` runs the MATLAB compiler to build and package the specified project. The installer is generated.

`productionServerCompiler -win32` instructs the compiler to build a 32-bit application on a 64-bit system when you use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.

Input Arguments

project_name - name of the project to be compiled

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

Purpose

Compile and package functions for external deployment

Syntax

`deploytool` [-win32] [[[-build] | [-project]]*`project_name`*]

Description

`deploytool` opens the MATLAB Compiler app.

`deploytool project_name` opens the MATLAB Compiler app with the project preloaded.

`deploytool -build project_name` runs the MATLAB Compiler to build the specified project. The installer is not generated.

`deploytool -package project_name` runs the MATLAB Compiler to build and package the specified project. The installer is generated.

`deploytool -win32` instructs the compiler to build a 32-bit application on a 64-bit system when the following are true:

- You use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.
- You are running from a Windows command line (not a MATLAB command line).

Input Arguments

project_name - name of the project to be compiled

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

Purpose

Compile MATLAB functions for deployment

Syntax

```
mcc {-e} | {-m} [-a filename]... [-B filename[:arg]...] [-C] [-d outFolder]
[-f filename] [-g] [-I directory]... [-K] [-M string] [-N] [-o filename]
[-p path]... [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename
```

```
mcc -l [-a filename]... [-B filename[:arg]...] [-C] [-d outFolder] [-f
filename] [-g] [-I directory]... [-K] [-M string] [-N] [-o filename]
[-p path]... [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename...
```

```
mcc -c [-a filename]... [-B filename[:arg]...] [-C] [-d outFolder] [-f
filename] [-g] [-I directory]... [-K] [-M string] [-N] [-o filename]
[-p path]... [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename...
```

```
mcc -W cpplib:component_name -T link:lib [-a filename]... [-B
filename[:arg]...] [-C] [-d outFolder] [-f filename] [-g] [-I directory]...
[-K] [-M string] [-N] [-o filename] [-p path]... [-R option] [-S] [-v] [-w
option[:msg]] [-win32] [-Y filename] mfilename...
```

```
mcc -W dotnet:component_name,[className], [framework_version],
security, remote_type -T link:lib [-a filename]... [-B filename[:arg]...]
[-C] [-d outFolder] [-f filename] [-I directory]... [-K] [-M string] [-N]
[-p path]... [-R option] [-S] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename... [class{className:[mfilename]...}]...
```

```
mcc -W excel:component_name,[className], [version] -T link:lib [-a
filename]... [-b] [-B filename[:arg]...] [-C] [-d outFolder] [-f filename]
[-I directory]... [-K] [-M string] [-N] [-p path]... [-R option] [-u] [-v]
[-w option[:msg]] [-win32] [-Y filename] mfilename...
```

```
mcc -W 'java:packageName,[className]' [-a filename]... [-b]
[-B filename[:arg]...] [-C] [-d outFolder] [-f filename] [-I
directory]... [-K] [-M string] [-N] [-p path]... [-R option]
[-S] [-v] [-w option[:msg]] [-win32] [-Y filenamem] filename...
[class{className:[mfilename]...}]...
```

`mcc` -W CTF:*`component_name`* [-a *`filename`*]... [-b] [-B *`filename`*[:*`arg`*]...] [-d *`outFolder`*] [-f *`filename`*] [-I *`directory`*]... [-K] [-M *`string`*] [-N] [-p *`path`*]... [-R *`option`*] [-S] [-v] [-w *`option`*[:*`msg`*]] [-win32] [-Y *`filename`*m] *`filename`*... [class{*`className`*:[m*`filename`*]...}]...

`mcc -?`

Description

`mcc -m mfilename` compiles the function into a standalone application.

This is equivalent to `-W main -T link:exe`.

`mcc -e mfilename` compiles the function into a standalone application that does not open an MS-DOS® command window.

This is equivalent to `-W WinMain -T link:exe`.

`mcc -l mfilename...` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

This is equivalent to `-W lib:`*`libname`* `-T link:lib`.

`mcc -c mfilename...` generates C wrapper code for the listed functions.

This is equivalent to `-W lib:`*`libname`* `-T codegen`.

`mcc -W cpplib:component_name -T link:lib mfilename...` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

`mcc -W dotnet:component_name,className,framework_version,security, remote_type -T link:lib mfilename...` creates a .NET component from the specified files.

- *component_name* — Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the .NET class to be created.
- *framework_version* — Specifies the version of the Microsoft .NET Framework you want to use to compile the component. Specify either:
  - `0.0` — Use the latest supported version on the target machine.
  - *version_major.version_minor* — Use a specific version of the framework.

  Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.
- *security* — Specifies whether the component to be created is a private assembly or a shared assembly.
  - To create a private assembly, specify `Private`.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- *remote_type* — Specifies the remoting type of the component. Values are `remote` and `local`.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{`*className*`:`*mfilename*`...}...`. *className* specifies the name of the class to create using *mfilename*.

`mcc -W excel:component_name,className, version -T link:lib mfilename...` creates a Microsoft Excel component from the specified files.

- *component_name* — Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`.

- *`className`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *`component_name`* as the default.
- *`version`* — Specifies the version of the component specified as *`major.minor`*.
  - *`major`* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
  - *`minor`* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.

`mcc -W 'java:packageName,className' mfilename...` creates a Java package from the specified files.

- *`packageName`* — Specifies the name of the Java package and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *`className`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *`packageName`*.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{`*`className`*`:`*`mfilename`*`...}...`. *`className`* specifies the name of the class to create using *`mfilename`*.

`mcc -W CTF:component_name` instructs the compiler to create a deployable CTF archive that is deployable in a MATLAB Production Server instance.

`mcc -?` displays help.

Tip You can issue the `mcc` command either from the MATLAB command prompt or the DOS or UNIX® command line.

Options

-a Add to Archive

Add a file to the CTF archive using

```
-a filename
```

to specify a file to be directly added to the CTF archive. Multiple `-a` options are permitted. MATLAB Compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files.

If only a folder name is included with the `-a` option, the entire contents of that folder are added recursively to the CTF archive. For example:

```
mcc -m hello.m -a ./testdir
```

In this example, `testdir` is a folder in the current working folder. All files in `testdir`, as well as all files in subfolders of `testdir`, are added to the CTF archive, and the folder subtree in `testdir` is preserved in the CTF archive.

If a wildcard pattern is included in the file name, only the files in the folder that match the pattern are added to the CTF archive and subfolders of the given path are not processed recursively. For example:

```
mcc -m hello.m -a ./testdir/*
```

In this example, all files in `./testdir` are added to the CTF archive and subfolders under `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

In this example, all files with the extension `.m` under `./testdir` are added to the CTF archive and subfolders of `./testdir` are not processed recursively.

All files added to the CTF archive using `-a` (including those that match a wildcard pattern or appear under a folder specified using `-a`) that do not appear on the MATLAB path at the time of compilation causes a path entry to be added to the deployed application's run-time path

so that they appear on the path when the deployed application or component executes.

When files are included, the absolute path for the DLL and header files is changed. The files are placed in the `.\exe_mcr\` folder when the CTF file is expanded. The file is not placed in the local folder. This folder is created from the CTF file the first time the EXE file is executed. The `isdeployed` function is provided to help you accommodate this difference in deployed mode.

The `-a` switch also creates a `.auth` file for authorization purposes. It ensures that the executable looks for the DLL- and H-files in the *exe*`_mcr\`*exe* folder.

Caution

If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

Note Currently, `*` is the only supported wildcard.

Note If the `-a` flag is used to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to update the `classpath` with the parent folder of the package.

-b Generate Excel Compatible Formula Function

Generate a Visual Basic® file (`.bas`) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function. This option requires MATLAB Builder™ EX.

-B Specify Bundle File

Replace the file on the `mcc` command line with the contents of the specified file. Use

```
-B filename[:<a1>,<a2>,...,<an>]
```

The bundle file `filename` should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file might contain other `-B` options. A bundle file can include replacement parameters for Compiler options that accept names and version numbers. See "Using Bundle Files to Build MATLAB Code" for a list of the bundle files included with MATLAB Compiler.

-C Do Not Embed CTF Archive by Default

Override automatically embedding the CTF archive in C/C++ and `main`/`Winmain` shared libraries and standalone binaries by default.

-d Specified Folder for Output

Place output in a specified folder. Use

```
-d outFolder
```

to direct the generated files to *`outFolder`*.

-f Specified Options File

Override the default options file with the specified options file. Use

```
-f filename
```

to specify `filename` as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of MATLAB Compiler. This option is a direct pass-through to `mbuild`.

-g Generate Debugging Information

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MCR, the function call, or the termination routine. This option does not let you debug your MATLAB files with a C/C++ debugger.

-G Debug Only

Same as `-g`.

-I Add Folder to Include Path

Add a new folder path to the list of included folders. Each `-I` option adds a folder to the beginning of the list of paths to search. For example,

```
-I <directory1> -I <directory2>
```

sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

-K Preserve Partial Output Files

Direct `mcc` to not delete output files if the compilation ends prematurely, due to error.

The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

-M Direct Pass Through

Define compile-time options. Use

```
-M string
```

to pass `string` directly to `mbuild`. This provides a useful mechanism for defining compile-time options, e.g., `-M "-Dmacro=value"`.

Note Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

-N Clear Path

Passing `-N` effectively clears the path of all folders except the following core folders (this list is subject to change over time):

- *`matlabroot`*`\toolbox\matlab`
- *`matlabroot`*`\toolbox\local`
- *`matlabroot`*`\toolbox\compiler\deploy`

It also retains all subfolders of the above list that appear on the MATLAB path at compile time. Including `-N` on the command line lets you replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the `-N` option retains all folders that you included on the path that are not under *`matlabroot`*`\toolbox`.

-o Specify Output Name

Specify the name of the final executable (standalone applications only). Use

```
-o outputfile
```

to name the final executable output of MATLAB Compiler. A suitable, possibly platform-dependent, extension is added to the specified name (e.g., `.exe` for Windows standalone applications).

-p Add Folder to Path

Use in conjunction with the required option `-N` to add specific folders (and subfolders) under *`matlabroot`*`\toolbox` to the compilation MATLAB path in an order sensitive way. Use the syntax

```
-N -p directory
```

where `directory` is the folder to be included. If `directory` is not an absolute path, it is assumed to be under the current working folder. The rules for how these folders are included follow.

- If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in an order-sensitive context.
- If a folder is included with `-p` that is not on the original MATLAB path, that folder is not included in the compilation. (You can use `-I` to add it.)

If a path is added with the `-I` option while this feature is active (`-N` has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with `-p`. Otherwise, the folder is added to the head of the path, as it normally would be with `-I`.

-R Run-Time

Provides MCR run-time options. The syntax is as follows:

```
-R option
```

Option	Description
`-logfile` *`filename`*	Specify a log file name.
`-nodisplay`	Suppress the MATLAB `nodisplay` run-time warning.
`-nojvm`	Do not use the Java Virtual Machine (JVM).

Option	Description
`-startmsg`	Customizable user message displayed at MCR initialization time.
`-completemsg`	Customizable user message displayed when MCR initialization is complete.

Note Not all `-R` options are available for all mcc targets.

Caution

When running on Mac OS X, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

-S Create Singleton MCR Context

The standard behavior for the MCR is that every instance of a class gets its own runtime context. This runtime context includes a global MATLAB workspace for variables such as the path and a base workspace for each function in the class. If multiple instances of a class are created, each instance gets an independent context. This ensures that changes made to the global, or base, workspace in one instance of the class does not effect other instances of the same class.

In a singleton MCR, all instances of a class share the runtime context. If multiple instances of a class are created, the use the runtime context created by the first instance. This saves start up time and some resources. However, any changes made to the global workspace or the base workspace by one instance impacts all of the class instances. For example, if `instance1` creates a global variable `A` in a singleton MCR, the `instance2` will be able to use variable `A`.

-T Specify Target Stage

Specify the output target phase and type.

Use the syntax `-T target` to define the output type. Target values are as follow.

Target	Description
`compile:exe`	Generate a C/C++ wrapper file plus compile C/C++ files to object form suitable for linking into a standalone application.
`compile:lib`	Generate a C/C++ wrapper file plus compile C/C++ files to object form suitable for linking into a shared library/DLL.
`link:exe`	Same as `compile:exe` plus links object files into a standalone application.
`link:lib`	Same as `compile:lib` plus links object files into a shared library/DLL.

-u Register COM Component for the Current User

Register COM component for the current user only on the development machine. The argument applies only for generic COM component and Microsoft Excel add-in targets only.

-v Verbose

Display the compilation steps, including:

- MATLAB Compiler version number
- The source file names as they are processed
- The names of the generated output files as they are created
- The invocation of `mbuild`

The `-v` option passes the `-v` option to `mbuild` and displays information about `mbuild`.

-w Warning Messages

Display warning messages. Use the syntax

```
-w option [:<msg>]
```

to control the display of warnings. This table lists the syntaxes.

Syntax	Description
`-w list`	Generate a table that maps `<string>` to warning message for use with `enable`, `disable`, and `error`. "Warning Messages", lists the same information.
`-w enable`	Enable complete warnings.
`-w disable[:<string>]`	Disable specific warnings associated with `<string>`. "Warning Messages", lists the `<string>` values. Omit the optional `<string>` to apply the `disable` action to all warnings.
`-w enable[:<string>]`	Enable specific warnings associated with `<string>`. "Warning Messages", lists the `<string>` values. Omit the optional `<string>` to apply the `enable` action to all warnings.
`-w error[:<string>]`	Treat specific warnings associated with `<string>` as an error. Omit the optional `<string>` to apply the `error` action to all warnings.

Syntax	Description
`-w off[:<string>] [<filename>]`	Turn warnings off for specific error messages defined by `<string>`. You can also narrow scope by specifying warnings be turned off when generated by specific `<filename>`s.
`-w on[:<string>] [<filename>]`	Turn warnings on for specific error messages defined by `<string>`. You can also narrow scope by specifying warnings be turned on when generated by specific `<filename>`s.

It is also possible to turn warnings on or off in your MATLAB code.

For example, to turn warnings off for deployed applications (specified using `isdeployed`) in your `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn warnings on for deployed applications, you write:

```
if isdeployed
    warning on
end
```

-win32 Run in 32-Bit Mode

Use this option to build a 32-bit application on a 64-bit system *only* when the following are true:

- You have a 32-bit installation of MATLAB.
- You use the same MATLAB installation root (`matlabroot`) for both 32-bit and 64-bit versions of MATLAB.

- You are running from a Windows command line.

-Y License File

Use

```
-Y license.lic
```

to override the default license file with the specified argument.